Concrete Error Handling Mechanisms Should Be Configurable

Tero Hasu
tero@ii.uib.no

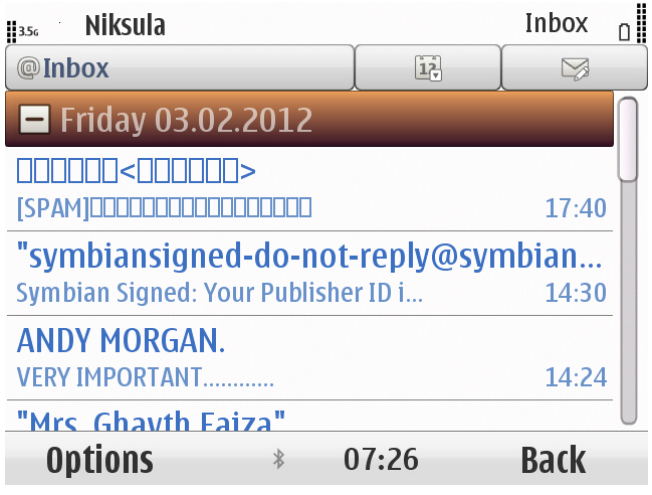
WEH 2012

9 June 2012

Mail Clients (in My Opinion)

- one of the best (2001–2008)

- one of the worst (2008–)



Where to Get a Better One for Current Symbian Phones?

- few (if any) third-party options for a “dead” niche platform
  - one that used to have a great built-in one
- large application
  - K-9 Mail (Android) $\approx$ 65000 LOC
- if make a new one, better at least make it portable

How to Make a Portable App?

- use C++, say, primarily—compilers are ubiquitous
- platform-specific GUI to drive it all
  - for native look and feel
- rest into subsystems, with cross-platform APIs, multiple implementations if need be:
  - contacts database integration (platform specific?)
  - persistence and search (portable?, with e.g. SQLite)
  - protocols and data formats (portable?)
  - socket transport (somewhat portable, but tricky)
  - etc.

What Kind of "Cross-Platform APIs"?

- use abstract types and associated operations
- but: multitasking and event delivery?
  - GUI probably driven by a specific event loop (GTK+, Qt, Symbian, UIKit, ...) $\rightarrow$ easiest to use that one
- but: error reporting and handling?
  - application frameworks have their own conventions $\rightarrow$ would be convenient for subsystem APIs to follow them
- events & errors: cross-cutting concerns
  - permeate the codebase
  - hard to abstract at library level

Consider: Portable/Adaptable Error Handling for C and C++

- focus on a specific problem first before generalizing
- may well be so commonplace as to warrant first-class language support
  - rather than solutions based on *AOP* or *model-driven engineering* or other generic approaches

Why C and C++?

- there's no single de facto standard error handling mechanism that everyone uses
- no error handling mechanism in the language for C at all
  - apart from `errno`, not much used beyond the C standard library
- C++ has exceptions, but there may be reasons not to use them

Why C++ without Exceptions?

- C++ often used together with C, and exception use makes language interoperability harder
- some C++ based toolchains do not support exceptions (e.g., CUDA, Symbian pre-v9)
- even when supported, it may be a platform convention not to use C++ exceptions (e.g., Bada)
- hard real time requirements may preclude their use
  - current C++ compilers do not give any timing guarantees w.r.t. exceptions

Vision: Have a Language-Level Error Handling Abstraction

- Do not decide on a specific error handling mechanism when implementing a C or C++ library.
- Make that decision at configuration time to adapt the library for its target context.
  - build context: adapt implementation, for toolchain and libraries
  - use context: adapt API, for application idioms
- For ease of library adoption: Vendor can specialize for different targets, and distribute as standard C or C++ code.

Requirements

- We might accept that when *using* a non-retargetable "legacy" C or C++ library internally within a codebase one would have to explicitly use whatever mechanism is exposed by the library API.
  - Manually convert to abstraction as required: E.g. Symbian *leaves* and C++ exceptions must not have overlapping extent
- But when *creating* a new library, one should have the option of using an abstraction.

To Language Creators

- If your compiler back end generates C or C++ source code and produces APIs in those languages for interoperability...
  - (our language, Magnolia, does...)
- ...consider providing choice of concrete error handling mechanisms to target in generated code
  - The Vala language demonstrates how to target C and GLib's GError as its (sole) concrete error handling mechanism.

Design and Implementation Challenges

- Choose abstraction and implement mapping different error propagation and resource cleanup mechanisms, which include:
  - conditionals, `gotos`, `returns`, exceptions, `setjmp`/`longjmp`, and libunwind for propagation; and
  - direct operations and declared cleanup actions (e.g., RAII, cleanup stack, autorelease pool) for cleanup.

Abstraction, Sketched

- here:
  - syntax inspired by C++ and Magnolia's alerts...
  - ...but focus on information content, not syntax

Abstraction: Declaring Kinds of Errors

```
alert { NoMemory, DbNoTable };
```

Abstraction: Defining Mappings to Concrete Errors

```
alertdef NoMemory
  // map to specified existing definitions
  in posix extern int ENOMEM,
  in cxx extern typename std::bad_alloc,
  in symbian extern TInt KErrNoMemory,
  in bada extern result E_OUT_OF_MEMORY;
alertdef DbNoTable
  // define by same name for all mechanisms
  default DbNoTable;
```

Abstraction: Declaring Whether Functions Can Fail

```
Db* dbOpen() guard;
void dbCreateTable(Db*) guard;
void dbExec(Db*, const char*) guard;
void dbClose(Db*);
```

- Must know which calls may fail so that can generate checks as necessary.

Abstraction: Raising an Error

```
Db* db = (Db*)malloc(sizeof(Db));
if (!db)
    raise NoMemory;
```

Abstraction: Handling Errors and Cleaning Up Resources

```
void execSql(const char* sql) {
  Db* db = dbOpen();
  // note C++11 lambda style "capture list"
  scope(exit) [&] dbClose(db);
 again:
  try { dbExec(db, sql); }
  rescue (DbNoTable _) {
    dbCreateTable(db);
    goto again;
  }
}
```

Concrete: errno Based

```
void execSql(const char* sql) {
  errno = 0; Db* db = dbOpen();
  if (errno) return;
 again:
  {
    errno = 0; dbExec(db, sql);
    if (errno == DbNoTable) {
      errno = 0; dbCreateTable(db);
      if (errno) goto __cleanup;
      goto again;
    }
    if (errno) goto __cleanup;
  }
 __cleanup:
  dbClose(db);
}
```

Concrete: GError Based

```
void execSql(const char* sql, GError **error) {
  Db* db = dbOpen(error);
  if (*error) return;
 again: {
    dbExec(db, sql, error);
    if (*error) {
      if (g_error_matches(*error, MAGNOLIA_ERROR,
                          DbNoTable)) {
        g_clear_error(error);
        dbCreateTable(db, error);
        if (*error) goto __cleanup;
        goto again;
      } else goto __cleanup;
    }
  }
 __cleanup: dbClose(db);
}
```

Concrete: GError Based, for GCC

```
void execSql(const char* sql, GError **error) {
  Db* db = dbOpen(error);
  if (*error) return;
  // nested function, tied to a dummy automatic
  void __dtor(int* __v) { dbClose(db); }
  int __dummy __attribute__((cleanup(__dtor)));
 again: {
    dbExec(db, sql, error);
    if (*error) {
      if (g_error_matches(*error, MAGNOLIA_ERROR,
                          DbNoTable)) {
        g_clear_error(error);
        dbCreateTable(db, error);
        if (*error) return;
        goto again;
      }
      return; }}}
```

Concrete: C++ Exceptions

```
struct __Dtor {
  Db*& db;
  __Dtor(Db*& db) : db(db) {}
  ~__Dtor() { dbClose(db); }
};

void execSql(const char* sql) {
  Db* db = dbOpen();
  __Dtor __dummy(db);
 again:
  {
    try { dbExec(db, sql); }
    catch (const DbNoTable& e) {
      dbCreateTable(db); goto again;
    }
  }
}
```

Concrete: C++ Exceptions, for C++11

```
void execSql(const char* sql) {
  Db* db = dbOpen();
  __Dtor __dummy([&] () { dbClose(db); });
 again:
  {
    try { dbExec(db, sql); }
    catch (const DbNoTable& e) {
      dbCreateTable(db); goto again;
    }
  }
}
```


Concrete: Symbian Leaves

```
struct T__Dtor { Db*& db;
  T__Dtor(Db*& db) : db(db) {}
  void Call() { dbClose(db); }
};
static void __f(TAny* aPtr) {
  (static_cast<T__Dtor*>(aPtr))->Call(); }

void execSqlL(const char* sql) {
  Db* db = dbOpenL();
  T__Dtor __dtor(db);
  CleanupStack::PushL(TCleanupItem(__f, &__dtor));
 again: {
    TRAPD(_e, dbExecL(db, sql));
    if (_e == DbNoTable) {
      dbCreateTableL(db); goto again;
    } else if (_e) User::Leave(_e);
  } CleanupStack::PopAndDestroy(); }
```

Conclusion

- there should be languages implemented by translation to C or C++
- with abstract error handling constructs
  - declare errors and their mappings to concrete mechanisms
  - declare whether a function can produce an error
  - raise an error
  - handle an error
  - install cleanup statements
- independent of concrete mechanisms
  - to enable translation to different mechanisms
    - to enable reuse